2.* The present implementation has a long ancestry, beginning in the spring of 1977, when its author wrote a prototype set of subroutines and macros that were used to develop the first Computer Modern fonts. This original proto-METAFONT required the user to recompile a SAIL program whenever any character was changed, because it was not a "language" for font design; the language was SAIL. After several hundred characters had been designed in that way, the author developed an interpretable language called METAFONT, in which it was possible to express the Computer Modern programs less cryptically. A complete METAFONT processor was designed and coded by the author in 1979. This program, written in SAIL, was adapted for use with a variety of typesetting equipment and display terminals by Leo Guibas, Lyle Ramshaw, and David Fuchs. Major improvements to the design of Computer Modern fonts were made in the spring of 1982, after which it became clear that a new language would better express the needs of letterform designers. Therefore an entirely new METAFONT language and system were developed in 1984; the present system retains the name and some of the spirit of METAFONT79, but all of the details have changed.

No doubt there still is plenty of room for improvement, but the author is firmly committed to keeping METAFONT84 "frozen" from now on; stability and reliability are to be its main virtues.

On the other hand, the WEB description can be extended without changing the core of METAFONT84 itself, and the program has been designed so that such extensions are not extremely difficult to make. The *banner* string defined here should be changed whenever METAFONT undergoes any modifications, so that it will be clear which version of METAFONT might be the guilty party when a problem arises.

If this program is changed, the resulting system should not be called 'METAFONT'; the official name 'METAFONT' by itself is reserved for software systems that are fully compatible with each other. A special test suite called the "TRAP test" is available for helping to determine whether an implementation deserves to be known as 'METAFONT' [cf. Stanford Computer Science report CS1095, January 1986].

define *banner* ≡ `'This␣is␣METAFONT,␣C␣Version␣2.7'` { printed when METAFONT starts }

7.* Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when METAFONT is being installed or when system wizards are fooling around with METAFONT without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the codewords '*debug* ... *gubed*', with apologies to people who wish to preserve the purity of English.

Similarly, there is some conditional code delimited by '**stat** ... **tats**' that is intended for use when statistics are to be kept about METAFONT's memory usage. The **stat** ... **tats** code also implements special diagnostic information that is printed when *tracingedges* > 1.

define *stat* ≡ *ifdef* (`'STAT'`)
define *tats* ≡ *endif* (`'STAT'`)
define *debug* ≡ *ifdef* (`'DEBUG'`)
define *gubed* ≡ *endif* (`'DEBUG'`)
format *stat* ≡ *begin*
format *tats* ≡ *end*

8.* This program has two important variations: (1) There is a long and slow version called INIMF, which does the extra calculations needed to initialize METAFONT's internal tables; and (2) there is a shorter and faster production version, which cuts the initialization to a bare minimum. Parts of the program that are needed in (1) but not in (2) are delimited by the codewords '**init** ... **tini**'.

define *init* ≡ *ifdef* (`'INIMF'`)
define *tini* ≡ *endif* (`'INIMF'`)
format *init* ≡ *begin*
format *tini* ≡ *end*

9:* If the first character of a Pascal comment is a dollar sign, Pascal-H treats the comment as a list of "compiler directives" that will affect the translation of this program into machine language. The directives shown below specify full checking and inclusion of the Pascal debugger when METAFONT is being debugged, but they cause range checking and other redundant code to be eliminated when the production system is being generated. Arithmetic overflow will be detected in all cases.

⟨ Compiler directives 9* ⟩ ≡
```
@{No compiler directives
for C.@}
```
This code is used in section 4.

11:* The following parameters can be changed at compile time to extend or reduce METAFONT's capacity. They may have different values in INIMF and in production versions of METAFONT.

define *file_name_size* ≡ *FILENAMESIZE* { Get value from `site.h`. }

⟨ Constants in the outer block 11* ⟩ ≡
mem_max = 262140; { greatest index in METAFONT's internal *mem* array; must be strictly less than *max_halfword*; must be equal to *mem_top* in INIMF, otherwise ≥ *mem_top* }
max_internal = 300; { maximum number of internal quantities }
buf_size = 3000; { maximum number of characters simultaneously present in current lines of open files; must not exceed *max_halfword* }
error_line = 79; { width of context lines on terminal error messages }
half_error_line = 50; { width of first lines of contexts in terminal error messages; should be between 30 and *error_line* − 15 }
max_print_line = 79; { width of longest text lines output; should be at least 60 }
screen_width = 1664; { number of pixels in each row of screen display }
screen_depth = 1200; { number of pixels in each column of screen display }
stack_size = 300; { maximum number of simultaneous input sources }
max_strings = 7500; { maximum number of strings; must not exceed *max_halfword* }
string_vacancies = 74000; { the minimum number of characters that should be available for the user's identifier names and strings, after METAFONT's own error messages are stored }
pool_size = 100000; { maximum number of characters in strings, including all error messages and help texts, and the names of all identifiers; must exceed *string_vacancies* by the total length of METAFONT's own strings, which is currently about 22000 }
move_size = 20000; { space for storing moves in a single octant }
max_wiggle = 1000; { number of autorounded points per cycle }
gf_buf_size = 16384; { size of the output buffer, must be a multiple of 8 }
pool_name = `'mf.pool'`; { string of length *file_name_size*; tells where the string pool appears }
path_size = 1000; { maximum number of knots between breakpoints of a path }
bistack_size = 1500; { size of stack for bisection algorithms; should probably be left at this value }
header_size = 100; { maximum number of TFM header words, times 4 }
lig_table_size = 15000;
{ maximum number of ligature/kern steps, must be at least 255 and at most 32510 }
max_kerns = 2500; { maximum number of distinct kern amounts }
max_font_dimen = 50; { maximum number of **fontdimen** parameters }
mem_top = 262140; { largest index in the *mem* array dumped by INIMF; must be substantially larger than *mem_min* and not greater than *mem_max* }
This code is used in section 4.

12.* Like the preceding parameters, the following quantities can be changed at compile time to extend or reduce METAFONT's capacity. But if they are changed, it is necessary to rerun the initialization program `INIMF` to generate new tables for the production METAFONT program. One can't simply make helter-skelter changes to the following constants, since certain rather complex initialization numbers are computed from them. They are defined here using `WEB` macros, instead of being put into Pascal's **const** list, in order to emphasize this distinction.

define *mem_min* = 0 { smallest index in the *mem* array, must not be less than *min_halfword* }
define *hash_size* = 9500
{ maximum number of symbolic tokens, must be less than *max_halfword* − 3 ∗ *param_size* }
define *hash_prime* = 7919 { a prime number equal to about 85% of *hash_size* }
define *max_in_open* = 15
{ maximum number of input files and error insertions that can be going on simultaneously }
define *param_size* = 150 { maximum number of simultaneous macro parameters }

16.* Here are some macros for common programming idioms.

define *negate*(#) ≡ # ← −# { change the sign of a variable }
define *double*(#) ≡ # ← # + # { multiply a variable by two }
define *loop* ≡ **while** *true* **do** { repeat over and over until a **goto** happens }
format *loop* ≡ *xclause* { `WEB`'s **xclause** acts like '**while** *true* **do**' }
define *do_nothing* ≡ { empty statement }
define *return* ≡ **goto** *exit* { terminate a procedure call }
format *return* ≡ *nil* { `WEB` will henceforth say **return** instead of *return* }

19:* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for font design; so the present specification of METAFONT has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes *'40* through *'176*; all of these characters are now available on most computer terminals.

Since we are dealing with more characters than were present in the first Pascal compilers, we have to decide what to call the associated data type. Some Pascals use the original name *char* for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters that are converted to and from *ASCII_code* when they are input and output. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

```
define text_char ≡ ASCII_code  { the data type of characters in text files }
define first_text_char = 0  { ordinal number of the smallest element of text_char }
define last_text_char = 255  { ordinal number of the largest element of text_char }
```

⟨ Local variables for initialization 19* ⟩ ≡

```
i: integer;
```

See also section 130.

This code is used in section 4.

22:* The ASCII code is "standard" only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. If METAFONT is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn't really matter what codes are specified in *xchr*[0 .. *'37*], but the safest policy is to blank everything out by using the code shown below.

However, other settings of *xchr* will make METAFONT more friendly on computers that have an extended character set, so that users can type things like '≠' instead of '<>'. People with extended character sets can assign codes arbitrarily, giving an *xchr* equivalent to whatever characters the users of METAFONT are allowed to have in their input files. Appropriate changes to METAFONT's *char_class* table should then be made. (Unlike TEX, each installation of METAFONT has a fixed assignment of category codes, called the *char_class*.) Such changes make portability of programs more difficult, so they should be introduced cautiously if at all.

```
define tab = '11  { ASCII horizontal tab }
define form_feed = '14  { ASCII form feed }
```

⟨ Set initial values of key variables 21 ⟩ +≡

```
for i ← 0 to '37 do xchr[i] ← chr(i);
for i ← '177 to '377 do xchr[i] ← chr(i);
```

24.* Input and output. The bane of portability is the fact that different operating systems treat input and output quite differently, perhaps because computer scientists have not given sufficient attention to this problem. People have felt somehow that input and output are not part of "real" programming. Well, it is true that some kinds of programming are more fun than others. With existing input/output conventions being so diverse and so messy, the only sources of joy in such parts of the code are the rare occasions when one can find a way to make the program a little less bad than it might have been. We have two choices, either to attack I/O now and get it over with, or to postpone I/O until near the end. Neither prospect is very attractive, so let's get it over with.

The basic operations we need to do are (1) inputting and outputting of text, to or from a file or the user's terminal; (2) inputting and outputting of eight-bit bytes, to or from a file; (3) instructing the operating system to initiate ("open") or to terminate ("close") input or output from a specified file; (4) testing whether the end of an input file has been reached; (5) display of bits on the user's screen. The bit-display operation will be discussed in a later section; we shall deal here only with more traditional kinds of I/O.

METAFONT needs to deal with two kinds of files. We shall use the term *alpha_file* for a file that contains textual data, and the term *byte_file* for a file that contains eight-bit binary information. These two types turn out to be the same on many computers, but sometimes there is a significant distinction, so we shall be careful to distinguish between them. Standard protocols for transferring such files from computer to computer, via high-speed networks, are now becoming available to more and more communities of users.

I/O in C is done using standard I/O. We will define the path numbers in an include file for C which are used in searching for files to be read. We'll define all the file types in C also.

⟨ Types in the outer block 18 ⟩ +≡
eight_bits = 0 .. 255; { unsigned one-byte quantity }

26.* All of the file opening functions are defined in C.

27.* And all the file closing routines as well.

30.* The *input_ln* function brings the next line of input from the specified field into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets *last* ← *first*. In general, the *ASCII_code* numbers that represent the next line of the file are input into *buffer*[*first*], *buffer*[*first* + 1], ..., *buffer*[*last* − 1]; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either *last* = *first* (in which case the line was entirely blank) or *buffer*[*last* − 1] ≠ "␣".

An overflow error is given, however, if the normal actions of *input_ln* would make *last* ≥ *buf_size*; this is done so that other parts of METAFONT can safely look at the contents of *buffer*[*last*+1] without overstepping the bounds of the *buffer* array. Upon entry to *input_ln*, the condition *first* < *buf_size* will always hold, so that there is always room for an "empty" line.

The variable *max_buf_stack*, which is used to keep track of how large the *buf_size* parameter must be to accommodate the present job, is also kept up to date by *input_ln*.

If the *bypass_eoln* parameter is *true*, *input_ln* will do a *get* before looking at the first character of the line; this skips over an *eoln* that was in *f*↑. The procedure does not do a *get* when it reaches the end of the line; therefore it can be used to acquire input from the user's terminal as well as from ordinary text files.

We define *input_ln* in C, for efficiency.

31.* The user's terminal acts essentially like other files of text, except that it is used both for input and for output. When the terminal is considered an input file, the file variable is called *term_in*, and when it is considered an output file the file variable is *term_out*.

define *term_in* ≡ *stdin* { the terminal as an input file }
define *term_out* ≡ *stdout* { the terminal as an output file }

32.* Here is how to open the terminal files. *t_open_out* does nothing. *t_open_in*, on the other hand, does the work of "rescanning," or getting any command line arguments the user has provided. It's defined in C.

```
define t_open_out ≡ { output already open for text output }
```

33.* Sometimes it is necessary to synchronize the input/output mixture that happens on the user's terminal, and three system-dependent procedures are used for this purpose. The first of these, *update_terminal*, is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent. The second, *clear_terminal*, is called when we wish to cancel any input that the user may have typed ahead (since we are about to issue an unexpected error message). The third, *wake_up_terminal*, is supposed to revive the terminal if the user has disabled it by some instruction to the operating system. The following macros show how these operations can be specified with UNIX. *update_terminal* does an *fflush* (via the macro *flush*), since that's easy. *wake_up_terminal* and *clear_terminal* are defined in C routines, if desired.

```
define update_terminal ≡ flush(term_out)
```

36.* The following program does the required initialization. Iff anything has been specified on the command line, then *t_open_in* will return with *last* > *first*.

```
function init_terminal: boolean; { gets the terminal input started }
  label exit;
  begin t_open_in;
  if last > first then
    begin loc ← first;
    while (loc < last) ∧ (buffer[loc] = ´␣´) do incr(loc);
    if loc < last then
      begin init_terminal ← true; goto exit;
      end;
    end;
  loop begin wake_up_terminal; write(term_out, ´**´); update_terminal;
    if ¬input_ln(term_in, true) then
      begin { this shouldn't happen }
      write_ln(term_out); write(term_out, ´!␣End␣of␣file␣on␣the␣terminal...␣why?´);
      init_terminal ← false; return;
      end;
    loc ← first;
    while (loc < last) ∧ (buffer[loc] = "␣") do incr(loc);
    if loc < last then
      begin init_terminal ← true; return; { return unless the line was all blank }
      end;
    write_ln(term_out, ´Please␣type␣the␣name␣of␣your␣input␣file.´);
    end;
exit: end;
```

51* **define** *bad_pool*(#) ≡
begin *wake_up_terminal*; *write_ln*(*term_out*, #); *a_close*(*pool_file*); *get_strings_started* ← *false*;
return;
end

⟨ Read the other strings from the MF.POOL file and return *true*, or give an error message and return *false* 51* ⟩ ≡
vstrcpy(*name_of_file* + 1, *pool_name*); { copy the string }
name_of_file[0] ← ´␣´; *name_of_file*[*strlen*(*pool_name*) + 1] ← ´␣´;
if *a_open_in*(*pool_file*, *MF_POOL_PATH*) **then**
begin *c* ← *false*;
repeat ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52* ⟩;
until *c*;
a_close(*pool_file*); *get_strings_started* ← *true*;
end
else *bad_pool*(´!␣I␣can´´t␣read␣mf.pool.´)
This code is used in section 47.

52* ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52* ⟩ ≡
begin if *eof*(*pool_file*) **then** *bad_pool*(´!␣mf.pool␣has␣no␣check␣sum.´);
read(*pool_file*, *m*); *read*(*pool_file*, *n*); { read two digits of string length }
if *m* = ´*´ **then** ⟨ Check the pool check sum 53* ⟩
else begin if (*xord*[*m*] < "0") ∨ (*xord*[*m*] > "9") ∨ (*xord*[*n*] < "0") ∨ (*xord*[*n*] > "9") **then**
bad_pool(´!␣mf.pool␣line␣doesn´´t␣begin␣with␣two␣digits.´);
l ← *xord*[*m*] ∗ 10 + *xord*[*n*] − "0" ∗ 11; { compute the length }
if *pool_ptr* + *l* + *string_vacancies* > *pool_size* **then** *bad_pool*(´!␣You␣have␣to␣increase␣POOLSIZE.´);
for *k* ← 1 **to** *l* **do**
begin if *eoln*(*pool_file*) **then** *m* ← ´␣´ **else** *read*(*pool_file*, *m*);
append_char(*xord*[*m*]);
end;
read_ln(*pool_file*); *g* ← *make_string*; *str_ref*[*g*] ← *max_str_ref*;
end;
end
This code is used in section 51*.

53* The WEB operation @$ denotes the value that should be at the end of this MF.POOL file; any other value means that the wrong pool file has been loaded.

⟨ Check the pool check sum 53* ⟩ ≡
begin *a* ← 0; *k* ← 1;
loop begin if (*xord*[*n*] < "0") ∨ (*xord*[*n*] > "9") **then**
bad_pool(´!␣mf.pool␣check␣sum␣doesn´´t␣have␣nine␣digits.´);
a ← 10 ∗ *a* + *xord*[*n*] − "0";
if *k* = 9 **then goto** *done*;
incr(*k*); *read*(*pool_file*, *n*);
end;
done: **if** *a* ≠ @$ **then** *bad_pool*(´!␣mf.pool␣doesn´´t␣match;␣tangle␣me␣again.´);
c ← *true*;
end
This code is used in section 52*.

61.* Here is the very first thing that METAFONT prints: a headline that identifies the version number and base name. The *term_offset* variable is temporarily incorrect, but the discrepancy is not serious since we assume that the banner and base identifier together will occupy at most *max_print_line* character positions.

```
⟨ Initialize the output routines 55 ⟩ +≡
  wterm(banner);
  if base_ident > 0 then print(base_ident);
  print_ln; update_terminal;
```

76:* The *jump_out* procedure just cuts across all active procedure levels and goes to *end_of_MF*. This is the only nontrivial **goto** statement in the whole program. It is used when there is no recovery from a particular error.

Some Pascal compilers do not implement non-local **goto** statements. In such cases the body of *jump_out* should simply be '*close_files_and_terminate*;' followed by a call on some system procedure that quietly terminates the program.

```
define do_final_end ≡
        begin update_terminal; ready_already ← 0;
        if (history ≠ spotless) ∧ (history ≠ warning_issued) then uexit(1)
        else uexit(0);
        end
```

```
⟨ Error handling procedures 73 ⟩ +≡
procedure jump_out;
  begin close_files_and_terminate; do_final_end;
  end;
```

79:* It is desirable to provide an '`E`' option here that gives the user an easy way to return from METAFONT to the system editor, with the offending line ready to be edited. We do this by calling the external procedure *call_edit* with a pointer to the filename, its length, and the line number. However, here we just set up the variables that will be used as arguments, since we don't want to do the switch-to-editor until after METAFONT has closed its files.

There is a secret '`D`' option available when the debugging routines have not been commented out.

```
define edit_file ≡ input_stack[file_ptr]
```

```
⟨ Interpret code c and return if done 79* ⟩ ≡
  case c of
  "0","1","2","3","4","5","6","7","8","9": if deletions_allowed then
      ⟨ Delete c − "0" tokens and goto continue 83 ⟩;
 debug"D": begin debug_help; goto continue; end; gubed
  "E": if file_ptr > 0 then
      begin edit_name_start ← str_start[edit_file.name_field];
      edit_name_length ← str_start[edit_file.name_field + 1] − str_start[edit_file.name_field];
      edit_line ← line; jump_out;
      end;
  "H": ⟨ Print the help information and goto continue 84 ⟩;
  "I": ⟨ Introduce new material from the terminal and return 82 ⟩;
  "Q","R","S": ⟨ Change the interaction level and return 81 ⟩;
  "X": begin interaction ← scroll_mode; jump_out;
    end;
  othercases do_nothing
  endcases;
  ⟨ Print the menu of available options 80 ⟩
```

This code is used in section 78.

153* Packed data. In order to make efficient use of storage space, METAFONT bases its major data structures on a *memory_word*, which contains either a (signed) integer, possibly scaled, or a small number of fields that are one half or one quarter of the size used for storing integers.

If x is a variable of type *memory_word*, it contains up to four fields that can be referred to as follows:

$x.int$	(an *integer*)
$x.sc$	(a *scaled* integer)
$x.hh.lh$, $x.hh.rh$	(two halfword fields)
$x.hh.b0$, $x.hh.b1$, $x.hh.rh$	(two quarterword fields, one halfword field)
$x.qqqq.b0$, $x.qqqq.b1$, $x.qqqq.b2$, $x.qqqq.b3$	(four quarterword fields)

This is somewhat cumbersome to write, and not very readable either, but macros will be used to make the notation shorter and more transparent. The Pascal code below gives a formal definition of *memory_word* and its subsidiary types, using packed variant records. METAFONT makes no assumptions about the relative positions of the fields within a word.

Since we are assuming 32-bit integers, a halfword must contain at least 16 bits, and a quarterword must contain at least 8 bits. But it doesn't hurt to have more bits; for example, with enough 36-bit words you might be able to have *mem_max* as large as 262142.

N.B.: Valuable memory space will be dreadfully wasted unless METAFONT is compiled by a Pascal that packs all of the *memory_word* variants into the space of a single integer. Some Pascal compilers will pack an integer whose subrange is '0 .. 255' into an eight-bit field, but others insist on allocating space for an additional sign bit; on such systems you can get 256 values into a quarterword only if the subrange is '−128 .. 127'.

The present implementation tries to accommodate as many variations as possible, so it makes few assumptions. If integers having the subrange '*min_quarterword* .. *max_quarterword*' can be packed into a quarterword, and if integers having the subrange '*min_halfword* .. *max_halfword*' can be packed into a halfword, everything should work satisfactorily.

It is usually most efficient to have *min_quarterword* = *min_halfword* = 0, so one should try to achieve this unless it causes a severe problem. The values defined here are recommended for most 32-bit computers.

define *min_quarterword* = 0 { smallest allowable value in a *quarterword* }
define *max_quarterword* = 255 { largest allowable value in a *quarterword* }
define *min_halfword* ≡ 0 { smallest allowable value in a *halfword* }
define *max_halfword* ≡ 262143 { largest allowable value in a *halfword* }

155* The operation of subtracting *min_halfword* occurs rather frequently in METAFONT, so it is convenient to abbreviate this operation by using the macro *ho* defined here. METAFONT will run faster with respect to compilers that don't optimize the expression '$x - 0$', if this macro is simplified in the obvious way when *min_halfword* = 0. Similarly, *qi* and *qo* are used for input to and output from quarterwords. So they have been simplified here in the obvious way.

define *ho*(#) ≡ #
define *qo*(#) ≡ #
define *qi*(#) ≡ #

156* The reader should study the following definitions closely:

define *sc* ≡ *int* { *scaled* data is equivalent to *integer* }

⟨ Types in the outer block 18 ⟩ +≡
quarterword = *min_quarterword* .. *max_quarterword*; { 1/4 of a word }
halfword = *min_halfword* .. *max_halfword*; { 1/2 of a word }
two_choices = 1 .. 2; { used when there are two variants in a record }
three_choices = 1 .. 3; { used when there are three variants in a record }

```
#include␣"memory.h";
```

169*. ⟨ Try to allocate within node p and its physical successors, and **goto** *found* if allocation was possible 169* ⟩ ≡

```
q ← p + node_size(p);  { find the physical successor }
while is_empty(q) do  { merge node p with node q }
  begin t ← rlink(q); tt ← llink(q);
  if q = rover then rover ← t;
  llink(t) ← tt; rlink(tt) ← t;
  q ← q + node_size(q);
  end;
r ← q − s;
if r > toint(p + 1) then ⟨ Allocate from the top of node p and goto found 170 ⟩;
if r = p then
  if rlink(p) ≠ p then ⟨ Allocate entire node p and goto found 171 ⟩;
node_size(p) ← q − p  { reset the size in case it grew }
```

This code is used in section 167.

178:* If METAFONT is extended improperly, the *mem* array might get screwed up. For example, some pointers might be wrong, or some "dead" nodes might not have been freed when the last reference to them disappeared. Procedures *check_mem* and *search_mem* are available to help diagnose such problems. These procedures make use of two arrays called *free* and *was_free* that are present only if METAFONT's debugging routines have been included. (You may want to decrease the size of *mem* while you are debugging.)

```
define free ≡ free_arr
⟨ Global variables 13 ⟩ +≡
debugfree: packed array [mem_min .. mem_max] of boolean; { free cells }
  was_free: packed array [mem_min .. mem_max] of boolean; { previously free cells }
  was_mem_end, was_lo_max, was_hi_min: pointer; { previous mem_end, lo_mem_max, and hi_mem_min }
  panicking: boolean; { do we want to check memory constantly? }
  gubed
```

182:* ⟨ Check variable-size *avail* list 182* ⟩ ≡

```
p ← rover; q ← null; clobbered ← false;
repeat if (p ≥ lo_mem_max) then clobbered ← true
  else if (rlink(p) ≥ lo_mem_max) then clobbered ← true
    else if ¬(is_empty(p)) ∨ (node_size(p) < 2) ∨ (p + node_size(p) > lo_mem_max) ∨
            (llink(rlink(p)) ≠ p) then clobbered ← true;
  if clobbered then
    begin print_nl("Double-AVAIL␣list␣clobbered␣at␣"); print_int(q); goto done2;
    end;
  for q ← p to p + node_size(p) − 1 do { mark all locations free }
    begin if free[q] then
      begin print_nl("Doubly␣free␣location␣at␣"); print_int(q); goto done2;
      end;
    free[q] ← true;
    end;
  q ← p; p ← rlink(p);
until p = rover;
done2:
```

This code is used in section 180.

194.* The following procedure, which is called just before METAFONT initializes its input and output, establishes the initial values of the date and time. It is calls an externally defined *date_and_time*, even though it could be done from Pascal. The external procedure also sets up interrupt catching.

Note that the values are *scaled* integers. Hence METAFONT can no longer be used after the year 32767.

```
procedure fix_date_and_time;
  begin date_and_time(internal[time], internal[day], internal[month], internal[year]);
  internal[time] ← internal[time] * unity; internal[day] ← internal[day] * unity;
  internal[month] ← internal[month] * unity; internal[year] ← internal[year] * unity;
  end;
```

199.* If changes are made to accommodate non-ASCII character sets, they should follow the guidelines in Appendix C of *The METAFONT book*.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  for k ← "0" to "9" do char_class[k] ← digit_class;
  char_class["."] ← period_class; char_class["␣"] ← space_class; char_class["%"] ← percent_class;
  char_class[""""] ← string_class;
  char_class[","] ← 5; char_class[";"] ← 6; char_class["("] ← 7; char_class[")"] ← right_paren_class;
  for k ← "A" to "Z" do char_class[k] ← letter_class;
  for k ← "a" to "z" do char_class[k] ← letter_class;
  char_class["_"] ← letter_class;
  char_class["<"] ← 10; char_class["="] ← 10; char_class[">"] ← 10; char_class[":"] ← 10;
  char_class["|"] ← 10;
  char_class["`"] ← 11; char_class["´"] ← 11;
  char_class["+"] ← 12; char_class["-"] ← 12;
  char_class["/"] ← 13; char_class["*"] ← 13; char_class["\"] ← 13;
  char_class["!"] ← 14; char_class["?"] ← 14;
  char_class["#"] ← 15; char_class["&"] ← 15; char_class["@"] ← 15; char_class["$"] ← 15;
  char_class["^"] ← 16; char_class["~"] ← 16;
  char_class["["] ← left_bracket_class; char_class["]"] ← right_bracket_class;
  char_class["{"] ← 19; char_class["}"] ← 19;
  for k ← 0 to "␣" − 1 do char_class[k] ← invalid_class;
  for k ← 127 to 255 do char_class[k] ← invalid_class;
  char_class[tab] ← space_class; char_class[form_feed] ← space_class;
```

564:* Online graphic output. METAFONT displays images on the user's screen by means of a few primitive operations that are defined below. These operations have deliberately been kept simple so that they can be implemented without great difficulty on a wide variety of machines. Since Pascal has no traditional standards for graphic output, some system-dependent code needs to be written in order to support this aspect of METAFONT; but the necessary routines are usually quite easy to write.

In fact, there are exactly four such routines:

init_screen does whatever initialization is necessary to support the other operations; it is a boolean function that returns *false* if graphic output cannot be supported (e.g., if the other three routines have not been written, or if the user doesn't have the right kind of terminal).

blank_rectangle updates a buffer area in memory so that all pixels in a specified rectangle will be set to the background color.

paint_row assigns values to specified pixels in a row of the buffer just mentioned, based on "transition" indices explained below.

update_screen displays the current screen buffer; the effects of *blank_rectangle* and *paint_row* commands may or may not become visible until the next *update_screen* operation is performed. (Thus, *update_screen* is analogous to *update_terminal*.)

The Pascal code here is a minimum version of *init_screen* and *update_screen*, usable on METAFONT installations that don't support screen output. If *init_screen* is changed to return *true* instead of *false*, the other routines will simply log the fact that they have been called; they won't really display anything. The standard test routines for METAFONT use this log information to check that METAFONT is working properly, but the *wlog* instructions should be removed from production versions of METAFONT.

These functions/procedures are defined externally in C.

567:* The *blank_rectangle* routine simply whitens all pixels that lie in columns *left_col* through *right_col* -1, inclusive, of rows *top_row* through *bot_row* -1, inclusive, given four parameters that satisfy the relations

$$0 \le left\_col \le right\_col \le screen\_width, \quad 0 \le top\_row \le bot\_row \le screen\_depth.$$

If *left_col* $=$ *right_col* or *top_row* $=$ *bot_row*, nothing happens.

The commented-out code in the following procedure is for illustrative purposes only.

Same thing

568:* The real work of screen display is done by *paint_row*. But it's not hard work, because the operation affects only one of the screen rows, and it affects only a contiguous set of columns in that row. There are four parameters: r (the row), b (the initial color), a (the array of transition specifications), and n (the number of transitions). The elements of a will satisfy

$$0 \le a[0] < a[1] < \cdots < a[n] \le screen\_width;$$

the value of r will satisfy $0 \le r < screen\_depth$; and n will be positive.

The general idea is to paint blocks of pixels in alternate colors; the precise details are best conveyed by means of a Pascal program (see the commented-out code below).

Same thing

768.* The file names we shall deal with for illustrative purposes have the following structure: If the name contains '/', the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains '.', the file extension consists of all such characters from the first remaining '.' to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters:

⟨ Global variables 13 ⟩ +≡
area_delimiter: *pool_pointer*; { the most recent '/', if any }
ext_delimiter: *pool_pointer*; { the most recent '.', if any }

769.* Input files that can't be found in the user's area may appear in a standard system area called *MF_area*. This system area name will, of course, vary from place to place.

In C, the default paths are specified in a separate file, `site.h`. The file opening procedures do path searching based either on those default paths, or on paths given by the user in environment variables.

771.* And here's the second.

```
function more_name(c : ASCII_code): boolean;
  begin if (c = " ") ∨ (c = tab) then more_name ← false
  else begin if (c = "/") then
      begin area_delimiter ← pool_ptr; ext_delimiter ← 0;
      end
    else if (c = ".") ∧ (ext_delimiter = 0) then ext_delimiter ← pool_ptr;
    str_room(1); append_char(c); { contribute c to the current string }
    more_name ← true;
    end;
  end;
```

775.* A messier routine is also needed, since base file names must be scanned before METAFONT's string mechanism has been initialized. We shall use the global variable *MF_base_default* to supply the text for default system areas and extensions related to base files.

In C, we don't give the area part, instead depending on the path searching that will happen during file opening. Also, the length will be set in the main program.

```
  define base_area_length = 0  { length of its area part }
  define base_ext_length = 5  { length of its '.base' part }
  define base_extension = ".base"  { the extension, as a WEB constant }
```

⟨ Global variables 13 ⟩ +≡
base_default_length: *integer*;
MF_base_default: *c_char_pointer*;

776.* We set the name of the default format file and the length of that name in C, instead of Pascal, since we want them to depend on the name of the program.

779:* Here is the only place we use *pack_buffered_name*. This part of the program becomes active when a "virgin" METAFONT is trying to get going, just after the preliminary initialization, or when the user is substituting another base file by typing '`&`' after the initial '`**`' prompt. The buffer contains the first line of input in *buffer*[*loc* .. (*last* − 1)], where *loc* < *last* and *buffer*[*loc*] ≠ "␣".

```
⟨ Declare the function called open_base_file 779* ⟩ ≡
function open_base_file: boolean;
  label found, exit;
  var j: 0 .. buf_size;  { the first space after the file name }
  begin j ← loc;
  if buffer[loc] = "&" then
    begin incr(loc); j ← loc; buffer[last] ← "␣";
    while buffer[j] ≠ "␣" do incr(j);
    pack_buffered_name(0, loc, j − 1);
    if w_open_in(base_file) then goto found;
    wake_up_terminal; wterm_ln('Sorry,␣I␣can''t␣find␣that␣base;', '␣will␣try␣the␣default.');
    update_terminal;
    end;  { now pull out all the stops: try for the system plain file }
  pack_buffered_name(base_default_length − base_ext_length, 1, 0);
  if ¬w_open_in(base_file) then
    begin wake_up_terminal; wterm_ln('I␣can''t␣find␣the␣default␣base␣file!');
    open_base_file ← false; return;
    end;
found: loc ← j; open_base_file ← true;
exit: end;
```

This code is used in section 1187.

781:* Now let's consider the "driver" routines by which METAFONT deals with file names in a system-independent manner. First comes a procedure that looks for a file name in the input by taking the information from the input buffer. (We can't use *get_next*, because the conversion to tokens would destroy necessary information.)

This procedure doesn't allow semicolons or percent signs to be part of file names, because of other conventions of METAFONT. The manual doesn't use semicolons or percents immediately after file names, but some users no doubt will find it natural to do so; therefore system-dependent changes to allow such characters in file names should probably be made with reluctance, and only when an entire file name that includes special characters is "quoted" somehow.

```
procedure scan_file_name;
  label done;
  begin begin_name;
  while (buffer[loc] = "␣") ∨ (buffer[loc] = tab) do incr(loc);
  loop begin if (buffer[loc] = ";") ∨ (buffer[loc] = "%") then goto done;
    if ¬more_name(buffer[loc]) then goto done;
    incr(loc);
    end;
done: end_name;
  end;
```

787* ⟨ Scan file name in the buffer 787* ⟩ ≡
```
begin begin_name; k ← first;
while ((buffer[k] = "␣") ∨ (buffer[k] = tab)) ∧ (k < last) do incr(k);
loop begin if k = last then goto done;
  if ¬more_name(buffer[k]) then goto done;
  incr(k);
  end;
done: end_name;
end
```
This code is used in section 786.

788* The *open_log_file* routine is used to open the transcript file and to help it catch up to what has previously been printed on the terminal.
```
procedure open_log_file;
var old_setting: 0 .. max_selector; { previous selector setting }
  k: 0 .. buf_size; { index into months and buffer }
  l: 0 .. buf_size; { end of first input line }
  m: integer; { the current month }
  months: c_char_pointer;
begin old_setting ← selector;
if job_name = 0 then job_name ← "mfput";
pack_job_name(".log");
while ¬a_open_out(log_file) do ⟨ Try to get a different log file name 789 ⟩;
log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
⟨ Print the banner line, including the date and time 790* ⟩;
input_stack[input_ptr] ← cur_input; { make sure bottom level is in memory }
print_nl("**"); l ← input_stack[0].limit_field − 1; { last position of first line }
for k ← 1 to l do print(buffer[k]);
print_ln; { now the transcript file contains the first line of input }
selector ← old_setting + 2; { log_only or term_and_log }
end;
```

790* ⟨ Print the banner line, including the date and time 790* ⟩ ≡
```
begin wlog(banner); print(base_ident); print("␣␣"); print_int(round_unscaled(internal[day]));
print_char("␣"); months ← ´␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
m ← round_unscaled(internal[month]);
for k ← 3 * m − 2 to 3 * m do wlog(months[k]);
print_char("␣"); print_int(round_unscaled(internal[year])); print_char("␣");
m ← round_unscaled(internal[time]); print_dd(m div 60); print_char(":"); print_dd(m mod 60);
end
```
This code is used in section 788*.

793:* Let's turn now to the procedure that is used to initiate file reading when an '`input`' command is being processed.

```
procedure start_input; { METAFONT will input something }
  label done;
  begin ⟨ Put the desired file name in (cur_name, cur_ext, cur_area) 795 ⟩;
  if cur_ext = "" then pack_file_name(cur_name, cur_area, ".mf")
  else pack_cur_name;
  loop begin begin_file_reading; { set up cur_file and new level of input }
    if a_open_in(cur_file, MF_INPUT_PATH) then goto done;
    if cur_ext = "" then { try to open the file without '.mf' }
      begin pack_cur_name;
      if a_open_in(cur_file, MF_INPUT_PATH) then goto done;
      end;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name("input␣file␣name", ".mf");
    end;
done: name ← a_make_name_string(cur_file); str_ref[cur_name] ← max_str_ref;
  if job_name = 0 then
    begin job_name ← cur_name; open_log_file;
    end; { open_log_file doesn't show_context, so limit and loc needn't be set to meaningful values yet }
  if term_offset + length(name) > max_print_line − 2 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  print_char("("); incr(open_parens); print(name); update_terminal;
  ⟨ Read the first line of the new file 794 ⟩;
  end;
```

965* ⟨ Shift the edges by (tx, ty), rounded 965* ⟩ ≡

```
tx ← round_unscaled(tx); ty ← round_unscaled(ty);
if (toint(m_min(cur_edges)) + tx ≤ 0) ∨ (m_max(cur_edges) + tx ≥ 8192) ∨
      (toint(n_min(cur_edges)) + ty ≤ 0) ∨ (n_max(cur_edges) + ty ≥ 8191) ∨
      (abs(tx) ≥ 4096) ∨ (abs(ty) ≥ 4096) then
  begin print_err("Too␣far␣to␣shift");
  help3("I␣can´t␣shift␣the␣picture␣as␣requested---it␣would")
  ("make␣some␣coordinates␣too␣large␣or␣too␣small.")
  ("Proceed,␣and␣I´ll␣omit␣the␣transformation."); put_get_error;
  end
else begin if tx ≠ 0 then
    begin if ¬valid_range(m_offset(cur_edges) − tx) then fix_offset;
    m_min(cur_edges) ← m_min(cur_edges) + tx; m_max(cur_edges) ← m_max(cur_edges) + tx;
    m_offset(cur_edges) ← m_offset(cur_edges) − tx; last_window_time(cur_edges) ← 0;
    end;
  if ty ≠ 0 then
    begin n_min(cur_edges) ← n_min(cur_edges) + ty; n_max(cur_edges) ← n_max(cur_edges) + ty;
    n_pos(cur_edges) ← n_pos(cur_edges) + ty; last_window_time(cur_edges) ← 0;
    end;
  end
```

This code is used in section 964.

1120.* The smallest d such that a given list can be covered with m intervals is determined by the *threshold* routine, which is sort of an inverse to *min_cover*. The idea is to increase the interval size rapidly until finding the range, then to go sequentially until the exact borderline has been discovered.

```
function threshold_fn(m : integer): scaled;
  var d: scaled; { lower bound on the smallest interval size }
  begin excess ← min_cover(0) − m;
  if excess ≤ 0 then threshold_fn ← 0
  else begin repeat d ← perturbation;
    until min_cover(d + d) ≤ m;
    while min_cover(d) > m do d ← perturbation;
    threshold_fn ← d;
    end;
  end;
```

1121.* The *skimp* procedure reduces the current list to at most m entries, by changing values if necessary. It also sets $info(p) \leftarrow k$ if $value(p)$ is the kth distinct value on the resulting list, and it sets *perturbation* to the maximum amount by which a *value* field has been changed. The size of the resulting list is returned as the value of *skimp*.

```
function skimp(m : integer): integer;
  var d: scaled; { the size of intervals being coalesced }
    p, q, r: pointer; { list manipulation registers }
    l: scaled; { the least value in the current interval }
    v: scaled; { a compromise value }
  begin d ← threshold_fn(m); perturbation ← 0; q ← temp_head; m ← 0; p ← link(temp_head);
  while p ≠ inf_val do
    begin incr(m); l ← value(p); info(p) ← m;
    if value(link(p)) ≤ l + d then ⟨ Replace an interval of values by its midpoint 1122 ⟩;
    q ← p; p ← link(p);
    end;
  skimp ← m;
  end;
```

1133.* Finally we're ready to actually write the `TFM` information. Here are some utility routines for this purpose.

Under UNIX, we are using the binary input and output routines. Hence, we redefine all the TFM input and output in terms of those routines.

```
define tfm_out(#) ≡ b_write_byte(tfm_file, #)
define tfm_two(#) ≡ b_write_2_bytes(tfm_file, #)
define tfm_four(#) ≡ b_write_4_bytes(tfm_file, #)
procedure tfm_qqqq(x : four_quarters); { output four quarterwords to tfm_file }
  begin tfm_out(qo(x.b0)); tfm_out(qo(x.b1)); tfm_out(qo(x.b2)); tfm_out(qo(x.b3));
  end;
```

1154.* The actual output of *gf_buf*[*a* .. *b*] to *gf_file* is performed by calling *write_gf*(*a*, *b*). It is safe to assume that *a* and *b* + 1 will both be multiples of 4 when *write_gf*(*a*, *b*) is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

In C, we use a macro to call *fwrite* or *write* directly, writing all the bytes to be written in one shot. Much better even than writing four bytes at a time.

1163.* Here is a routine that gets a GF file off to a good start.

```
define check_gf ≡ if output_file_name = 0 then init_gf
⟨ Declare generic font output procedures 1155 ⟩ +≡
procedure init_gf;
  var k: 0 .. 256;  { runs through all possible character codes }
    t: integer;  { the time of this run }
  begin gf_min_m ← 4096; gf_max_m ← −4096; gf_min_n ← 4096; gf_max_n ← −4096;
  for k ← 0 to 255 do char_ptr[k] ← −1;
  ⟨ Determine the file extension, gf_ext 1164 ⟩;
  set_output_file_name; gf_out(pre); gf_out(gf_id_byte);  { begin to output the preamble }
  old_setting ← selector; selector ← new_string; print("␣METAFONT␣output␣");
  print_int(round_unscaled(internal[year])); print_char("."); print_dd(round_unscaled(internal[month]));
  print_char("."); print_dd(round_unscaled(internal[day])); print_char(":");
  t ← round_unscaled(internal[time]); print_dd(t div 60); print_dd(t mod 60);
  selector ← old_setting; gf_out(cur_length); str_start[str_ptr + 1] ← pool_ptr; gf_string(0, str_ptr);
  pool_ptr ← str_start[str_ptr];  { flush that string from memory }
  gf_prev_ptr ← gf_offset + gf_ptr;
  end;
```

1169.* In this loop, *prev_w* represents the weight at column *prev_m*, which is the most recent column reflected in the output so far; *w* represents the weight at column *m*, which is the most recent column in the edge data. Several edges might cancel at the same column position, so we need to look ahead to column *mm* before actually outputting anything.

```
⟨ Output the pixels of edge row p to font row n 1169* ⟩ ≡
  if unsorted(p) > void then sort_edges(p);
  q ← sorted(p); w ← 0; prev_m ← −fraction_one;  { fraction_one ≈ ∞ }
  ww ← 0; prev_w ← 0; m ← prev_m;
  repeat if q = sentinel then mm ← fraction_one
    else begin d ← ho(info(q)); mm ← d div 8; ww ← ww + (d mod 8) − zero_w;
      end;
    if mm ≠ m then
      begin if prev_w ≤ 0 then
        begin if w > 0 then ⟨ Start black at (m, n) 1170 ⟩;
        end
      else if w ≤ 0 then ⟨ Stop black at (m, n) 1171 ⟩;
      m ← mm;
      end;
    w ← ww; q ← link(q);
  until mm = fraction_one;
  if w ≠ 0 then  { this should be impossible }
    print_nl("(There´s␣unbounded␣black␣in␣character␣shipped␣out!)");
  if prev_m − toint(m_offset(cur_edges)) + x_off > gf_max_m then
    gf_max_m ← prev_m − m_offset(cur_edges) + x_off
```

This code is used in section 1167.

1188.* Base files consist of *memory_word* items, and we use the following macros to dump words of different types:

⟨ Global variables 13 ⟩ +≡
base_file: *word_file*; { for input or output of base information }

1189.* The inverse macros are slightly more complicated, since we need to check the range of the values we are reading in. We say '*undump*(*a*)(*b*)(*x*)' to read an integer value x that is supposed to be in the range $a \le x \le b$.

```
define undump_end_end(#) ≡ # ← x; end
define undump_end(#) ≡ (x > #) then goto off_base else undump_end_end
define undump(#) ≡
        begin undump_int(x);
        if (x < #) ∨ undump_end
define undump_size_end_end(#) ≡ too_small(#) else undump_end_end
define undump_size_end(#) ≡
          if x > # then undump_size_end_end
define undump_size(#) ≡
        begin undump_int(x);
        if x < # then goto off_base;
        undump_size_end
```

1191.* Sections of a WEB program that are "commented out" still contribute strings to the string pool; therefore INIMF and METAFONT will have the same strings. (And it is, of course, a good thing that they do.)

⟨ Undump constants for consistency check 1191* ⟩ ≡

```
undump_int(x);
if x ≠ @$ then goto off_base; { check that strings are the same }
undump_int(x);
if x ≠ mem_min then goto off_base;
undump_int(x);
if x ≠ mem_top then goto off_base;
undump_int(x);
if x ≠ hash_size then goto off_base;
undump_int(x);
if x ≠ hash_prime then goto off_base;
undump_int(x);
if x ≠ max_in_open then goto off_base
```

This code is used in section 1187.

1199.* ⟨ Undump a few more things and the closing check word 1199* ⟩ ≡

```
undump(max_given_internal)(max_internal)(int_ptr);
for k ← 1 to int_ptr do
  begin undump_int(internal[k]); undump(0)(str_ptr)(int_name[k]);
  end;
undump(0)(frozen_inaccessible)(start_sym); undump(batch_mode)(error_stop_mode)(interaction);
undump(0)(str_ptr)(base_ident); undump(1)(hash_end)(bg_loc); undump(1)(hash_end)(eg_loc);
undump_int(serial_no);
undump_int(x); if (x ≠ 69069) ∨ feof(base_file) then goto off_base
```

This code is used in section 1187.

1200.* ⟨ Create the *base_ident*, open the base file, and inform the user that dumping has begun 1200* ⟩ ≡
selector ← *new_string*; *print*("␣(base="); *print*(*job_name*); *print_char*("␣");
print_int(*round_unscaled*(*internal*[*year*]) **mod** 100); *print_char*(".");
print_int(*round_unscaled*(*internal*[*month*])); *print_char*("."); *print_int*(*round_unscaled*(*internal*[*day*]));
print_char(")");
if *interaction* = *batch_mode* **then** *selector* ← *log_only*
else *selector* ← *term_and_log*;
str_room(1); *base_ident* ← *make_string*; *str_ref*[*base_ident*] ← *max_str_ref*;
pack_job_name(*base_extension*);
while ¬*w_open_out*(*base_file*) **do** *prompt_file_name*("base␣file␣name", *base_extension*);
print_nl("Beginning␣to␣dump␣on␣file␣"); *print*(*w_make_name_string*(*base_file*));
flush_string(*str_ptr* − 1); *print_nl*(*base_ident*)

This code is used in section 1186.

1204.* Now this is really it: METAFONT starts and ends here.

The initial test involving *ready_already* should be deleted if the Pascal runtime system is smart enough to detect such a "mistake."

```
  begin { start_here }
  history ← fatal_error_stop; { in case we quit during initialization }
  t_open_out; { open the terminal for output }
  set_paths(MF_BASE_PATH_BIT + MF_INPUT_PATH_BIT + MF_POOL_PATH_BIT);
  if ready_already = 314159 then goto start_of_MF;
  ⟨ Check the "constant" values for consistency 14 ⟩
  if bad > 0 then
    begin wterm_ln('Ouch---my␣internal␣constants␣have␣been␣clobbered!', '---case␣', bad : 1);
    goto final_end;
    end;
  initialize; { set global variables to their starting values }
  init if ¬get_strings_started then goto final_end;
  init_tab; { initialize the tables }
  init_prim; { call primitive for each primitive }
  init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr;
  max_str_ptr ← str_ptr; max_pool_ptr ← pool_ptr; fix_date_and_time;
  tini
  ready_already ← 314159;
start_of_MF: ⟨ Initialize the output routines 55 ⟩;
  ⟨ Get the first line of input and prepare to start 1211 ⟩;
  history ← spotless; { ready to go! }
  if start_sym > 0 then { insert the 'everyjob' symbol }
    begin cur_sym ← start_sym; back_input;
    end;
  main_control; { come to life }
  final_cleanup; { prepare for death }
  close_files_and_terminate;
final_end: do_final_end;
  end.
```

1205.* Here we do whatever is needed to complete METAFONT's job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of "safe" operations that cannot produce error messages. For example, it would be a mistake to call *str_room* or *make_string* at this time, because a call on *overflow* might lead to an infinite loop.

This program doesn't bother to close the input files that may still be open.

```
⟨ Last-minute procedures 1205* ⟩ ≡
procedure close_files_and_terminate;
  var k: integer;  { all-purpose index }
    lh: integer;  { the length of the TFM header, in words }
    lk_offset: 0 .. 256;  { extra words inserted at beginning of lig_kern array }
    p: pointer;  { runs through a list of TFM dimensions }
    x: scaled;  { a tfm_width value being output to the GF file }
  begin stat if internal[tracing_stats] > 0 then ⟨ Output statistics about this job 1208 ⟩; tats
  wake_up_terminal; ⟨ Finish the TFM and GF files 1206 ⟩;
  if log_opened then
    begin wlog_cr; a_close(log_file); selector ← selector − 2;
    if selector = term_only then
      begin print_nl("Transcript␣written␣on␣"); print(log_name); print_char(".");
      end;
    end;
  print_ln;
  if (edit_name_start ≠ 0) ∧ (interaction > batch_mode) then
    call_edit(str_pool, edit_name_start, edit_name_length, edit_line);
  end;
```

See also sections 1209, 1210, and 1212.

This code is used in section 1202.

1214.* System-dependent changes. Here are the variables used to hold "switch-to-editor" information.

⟨ Global variables 13 ⟩ +≡
edit_name_start: *pool_pointer*;
edit_name_length, *edit_line*: *integer*;

1215.* The *edit_name_start* will be set to point into *str_pool* somewhere after its beginning if METAFONT is supposed to switch to an editor on exit.

⟨ Set initial values of key variables 21 ⟩ +≡
edit_name_start ← 0;

1216.* Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. *All references are to section numbers instead of page numbers.*

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing METAFONT in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 25 sections are listed under "inner loop"; these account for more than 60% of METAFONT's running time, exclusive of input and output.

The following sections were changed by the change file: 2, 7, 8, 9, 11, 12, 16, 19, 22, 24, 26, 27, 30, 31, 32, 33, 36, 51, 52, 53, 61, 76, 79, 153, 155, 156, 169, 178, 182, 194, 199, 564, 567, 568, 768, 769, 771, 775, 776, 779, 781, 787, 788, 790, 793, 965, 1120, 1121, 1133, 1154, 1163, 1169, 1188, 1189, 1191, 1199, 1200, 1204, 1205, 1214, 1215, 1216.

⟨ Declare the procedure called *macro_call* 720 ⟩ Used in section 706.
⟨ Declare the procedure called *make_eq* 1001 ⟩ Used in section 995.
⟨ Declare the procedure called *make_exp_copy* 855 ⟩ Used in section 651.
⟨ Declare the procedure called *print_arg* 723 ⟩ Used in section 720.
⟨ Declare the procedure called *print_cmd_mod* 625 ⟩ Used in section 227.
⟨ Declare the procedure called *print_dp* 805 ⟩ Used in section 801.
⟨ Declare the procedure called *print_macro_name* 722 ⟩ Used in section 720.
⟨ Declare the procedure called *print_weight* 333 ⟩ Used in section 332.
⟨ Declare the procedure called *runaway* 665 ⟩ Used in section 162.
⟨ Declare the procedure called *scan_text_arg* 730 ⟩ Used in section 720.
⟨ Declare the procedure called *show_token_list* 217 ⟩ Used in section 162.
⟨ Declare the procedure called *skew_line_edges* 510 ⟩ Used in section 506.
⟨ Declare the procedure called *solve_choices* 284 ⟩ Used in section 269.
⟨ Declare the procedure called *split_cubic* 410 ⟩ Used in section 406.
⟨ Declare the procedure called *try_eq* 1006 ⟩ Used in section 995.
⟨ Declare the recycling subroutines 268, 385, 487, 620, 809 ⟩ Used in section 246.
⟨ Declare the stashing/unstashing routines 799, 800 ⟩ Used in section 801.
⟨ Declare unary action procedures 899, 900, 901, 904, 908, 910, 913, 916, 919 ⟩ Used in section 898.
⟨ Decrease the string reference count, if the current token is a string 743 ⟩
Used in sections 83, 742, 991, and 1016.
⟨ Decrease the velocities, if necessary, to stay inside the bounding triangle 300 ⟩ Used in section 299.
⟨ Decrease k by 1, maintaining the invariant relations between x, y, and q 123 ⟩ Used in section 121.
⟨ Decry the invalid character and **goto** *restart* 670 ⟩ Used in section 669.
⟨ Decry the missing string delimiter and **goto** *restart* 672 ⟩ Used in section 671.
⟨ Define an extensible recipe 1113 ⟩ Used in section 1106.
⟨ Delete all the row headers 353 ⟩ Used in section 352.
⟨ Delete empty rows at the top and/or bottom; update the boundary values in the header 352 ⟩
Used in section 348.
⟨ Delete $c -$ "0" tokens and **goto** *continue* 83 ⟩ Used in section 79*.
⟨ Descend one level for the attribute *info*(t) 245 ⟩ Used in section 242.
⟨ Descend one level for the subscript *value*(t) 244 ⟩ Used in section 242.
⟨ Descend past a collective subscript 1012 ⟩ Used in section 1011.
⟨ Descend the structure 1047 ⟩ Used in section 1046.
⟨ Descend to the previous level and **goto** *not_found* 561 ⟩ Used in section 560.
⟨ Determine if a character has been shipped out 1181 ⟩ Used in section 906.
⟨ Determine the before-and-after values of both coordinates 445 ⟩ Used in sections 444 and 446.
⟨ Determine the dependency list s to substitute for the independent variable p 816 ⟩ Used in section 815.
⟨ Determine the envelope's starting and ending lattice points $(m0, n0)$ and $(m1, n1)$ 508 ⟩
Used in section 506.
⟨ Determine the file extension, *gf_ext* 1164 ⟩ Used in section 1163*.
⟨ Determine the number n of arguments already supplied, and set *tail* to the tail of *arg_list* 724 ⟩
Used in section 720.
⟨ Determine the octant boundary q that precedes f 400 ⟩ Used in section 398.
⟨ Determine the octant code for direction (dx, dy) 480 ⟩ Used in section 479.
⟨ Determine the path join parameters; but **goto** *finish_path* if there's only a direction specifier 874 ⟩
Used in section 869.
⟨ Determine the starting and ending lattice points $(m0, n0)$ and $(m1, n1)$ 467 ⟩ Used in section 465.
⟨ Determine the tension and/or control points 881 ⟩ Used in section 874.
⟨ Dispense with the cases $a < 0$ and/or $b > l$ 979 ⟩ Used in section 978.
⟨ Display a big node 803 ⟩ Used in section 802.
⟨ Display a collective subscript 221 ⟩ Used in section 218.
⟨ Display a complex type 804 ⟩ Used in section 802.

⟨ Log the subfile sizes of the TFM file 1141 ⟩ Used in section 1134.
⟨ Make a special knot node for **pencircle** 896 ⟩ Used in section 895.
⟨ Make a trivial one-point path cycle 1066 ⟩ Used in section 1065.
⟨ Make moves for current subinterval; if bisection is necessary, push the second subinterval onto the stack, and **goto** *continue* in order to handle the first subinterval 314 ⟩ Used in section 311.
⟨ Make one move of each kind 317 ⟩ Used in section 314.
⟨ Make sure that all the diagonal roundings are safe 446 ⟩ Used in section 444.
⟨ Make sure that both nodes p and pp are of *structured* type 243 ⟩ Used in section 242.
⟨ Make sure that both x and y parts of p are known; copy them into *cur_x* and *cur_y* 873 ⟩ Used in section 872.
⟨ Make sure that the current expression is a valid tension setting 883 ⟩ Used in sections 882 and 882.
⟨ Make the dynamic memory into one big available node 1207 ⟩ Used in section 1206.
⟨ Make the envelope moves for the current octant and insert them in the pixel data 512 ⟩ Used in section 506.
⟨ Make the first 256 strings 48 ⟩ Used in section 47.
⟨ Make the moves for the current octant 468 ⟩ Used in section 465.
⟨ Make variable $q + s$ newly independent 586 ⟩ Used in section 232.
⟨ Massage the TFM heights, depths, and italic corrections 1126 ⟩ Used in section 1206.
⟨ Massage the TFM widths 1124 ⟩ Used in section 1206.
⟨ Merge row pp into row p 368 ⟩ Used in section 366.
⟨ Merge the *temp_head* list into *sorted*(h) 347 ⟩ Used in section 346.
⟨ Move right then up 319 ⟩ Used in sections 317 and 317.
⟨ Move the dependent variable p into both parts of the pair node r 947 ⟩ Used in section 946.
⟨ Move to next line of file, or **goto** *restart* if there is no next line 679 ⟩ Used in section 669.
⟨ Move to row *n0*, pointed to by p 377 ⟩ Used in sections 375, 376, 381, 382, 383, and 384.
⟨ Move to the next remaining triple (p, q, r), removing and skipping past zero-length lines that might be present; **goto** *done* if all triples have been processed 532 ⟩ Used in section 531.
⟨ Move to the right m steps 316 ⟩ Used in section 314.
⟨ Move up then right 320 ⟩ Used in sections 317 and 317.
⟨ Move upward n steps 315 ⟩ Used in section 314.
⟨ Multiply when at least one operand is known 942 ⟩ Used in section 941.
⟨ Multiply y by $\exp(-z/2^{27})$ 136 ⟩ Used in section 135.
⟨ Negate the current expression 903 ⟩ Used in section 898.
⟨ Normalize the given direction for better accuracy; but **return** with zero result if it's zero 540 ⟩ Used in section 539.
⟨ Numbered cases for *debug_help* 1213 ⟩ Used in section 1212.
⟨ Other local variables for *disp_edges* 580 ⟩ Used in section 577.
⟨ Other local variables for *fill_envelope* 511 ⟩ Used in sections 506 and 518.
⟨ Other local variables for *find_direction_time* 542 ⟩ Used in section 539.
⟨ Other local variables for *make_choices* 280 ⟩ Used in section 269.
⟨ Other local variables for *make_spec* 453 ⟩ Used in section 402.
⟨ Other local variables for *offset_prep* 495 ⟩ Used in section 491.
⟨ Other local variables for *scan_primary* 831, 836, 843 ⟩ Used in section 823.
⟨ Other local variables for *solve_choices* 286 ⟩ Used in section 284.
⟨ Other local variables for *xy_swap_edges* 357, 363 ⟩ Used in section 354.
⟨ Output statistics about this job 1208 ⟩ Used in section 1205*.
⟨ Output the answer, v (which might have become *known*) 934 ⟩ Used in section 932.
⟨ Output the character information bytes, then output the dimensions themselves 1136 ⟩ Used in section 1134.
⟨ Output the character represented in *cur_edges* 1167 ⟩ Used in section 1165.
⟨ Output the extensible character recipes and the font metric parameters 1140 ⟩ Used in section 1134.
⟨ Output the ligature/kern program 1139 ⟩ Used in section 1134.
⟨ Output the pixels of edge row p to font row n 1169* ⟩ Used in section 1167.
⟨ Output the subfile sizes and header bytes 1135 ⟩ Used in section 1134.

⟨ Read the first line of the new file 794 ⟩ Used in section 793*.
⟨ Read the other strings from the MF.POOL file and return *true*, or give an error message and return *false* 51* ⟩ Used in section 47.
⟨ Record a label in a lig/kern subprogram and **goto** *continue* 1111 ⟩ Used in section 1107.
⟨ Record a line segment from (xx, yy) to (xp, yp) dually in *env_move* 522 ⟩ Used in section 521.
⟨ Record a line segment from (xx, yy) to (xp, yp) in *env_move* 516 ⟩ Used in section 515.
⟨ Record a new maximum coefficient of type t 814 ⟩ Used in section 812.
⟨ Record a possible transition in column m 583 ⟩ Used in section 582.
⟨ Recycle a big node 810 ⟩ Used in section 809.
⟨ Recycle a dependency list 811 ⟩ Used in section 809.
⟨ Recycle an independent variable 812 ⟩ Used in section 809.
⟨ Recycle any sidestepped *independent* capsules 925 ⟩ Used in section 922.
⟨ Reduce comparison of big nodes to comparison of scalars 939 ⟩ Used in section 936.
⟨ Reduce to simple case of straight line and **return** 302 ⟩ Used in section 285.
⟨ Reduce to simple case of two givens and **return** 301 ⟩ Used in section 285.
⟨ Reduce to the case that $a, c \ge 0$, $b, d > 0$ 118 ⟩ Used in section 117.
⟨ Reduce to the case that $f \ge 0$ and $q > 0$ 110 ⟩ Used in sections 109 and 112.
⟨ Reflect the edge-and-weight data in *sorted*(p) 339 ⟩ Used in section 337.
⟨ Reflect the edge-and-weight data in *unsorted*(p) 338 ⟩ Used in section 337.
⟨ Remove a subproblem for *make_moves* from the stack 312 ⟩ Used in section 311.
⟨ Remove dead cubics 447 ⟩ Used in section 402.
⟨ Remove the left operand from its container, negate it, and put it into dependency list p with constant term q 1007 ⟩ Used in section 1006.
⟨ Remove the line from p to q, and adjust vertex q to introduce a new line 534 ⟩ Used in section 531.
⟨ Remove *open* types at the breakpoints 282 ⟩ Used in section 278.
⟨ Repeat a loop 712 ⟩ Used in section 707.
⟨ Replace an interval of values by its midpoint 1122 ⟩ Used in section 1121*.
⟨ Replace a by an approximation to $\sqrt{a^2 + b^2}$ 125 ⟩ Used in section 124.
⟨ Replace a by an approximation to $\sqrt{a^2 - b^2}$ 127 ⟩ Used in section 126.
⟨ Replicate every row exactly s times 341 ⟩ Used in section 340.
⟨ Report an unexpected problem during the choice-making 270 ⟩ Used in section 269.
⟨ Report overflow of the input buffer, and abort 34 ⟩
⟨ Report redundant or inconsistent equation and **goto** *done* 1004 ⟩ Used in section 1003.
⟨ Return an appropriate answer based on z and *octant* 141 ⟩ Used in section 139.
⟨ Revise the values of α, β, γ, if necessary, so that degenerate lines of length zero will not be obtained 529 ⟩ Used in section 528.
⟨ Rotate the cubic between p and q; then **goto** *found* if the rotated cubic travels due east at some time tt; but **goto** *not_found* if an entire cyclic path has been traversed 541 ⟩ Used in section 539.
⟨ Run through the dependency list for variable t, fixing all nodes, and ending with final link q 605 ⟩ Used in section 604.
⟨ Save string *cur_exp* as the *err_help* 1083 ⟩ Used in section 1082.
⟨ Scale the x coordinates of each row by s 343 ⟩ Used in section 342.
⟨ Scale the edges, shift them, and **return** 964 ⟩ Used in section 963.
⟨ Scale up *del1*, *del2*, and *del3* for greater accuracy; also set *del* to the first nonzero element of (*del1*, *del2*, *del3*) 408 ⟩ Used in sections 407, 413, and 420.
⟨ Scan a binary operation with '**of**' between its operands 839 ⟩ Used in section 823.
⟨ Scan a bracketed subscript and set *cur_cmd* ← *numeric_token* 861 ⟩ Used in section 860.
⟨ Scan a curl specification 876 ⟩ Used in section 875.
⟨ Scan a delimited primary 826 ⟩ Used in section 823.
⟨ Scan a given direction 877 ⟩ Used in section 875.
⟨ Scan a grouped primary 832 ⟩ Used in section 823.
⟨ Scan a mediation construction 859 ⟩ Used in section 823.

⟨ Skip down *prev_n* − *n* rows 1174 ⟩ Used in section 1172.
⟨ Skip to **elseif** or **else** or **fi**, then **goto** *done* 749 ⟩ Used in section 748.
⟨ Skip to column *m* in the next row and **goto** *done*, or skip zero rows 1173 ⟩ Used in section 1172.
⟨ Sort *p* into the list starting at *rover* and advance *p* to *rlink*(*p*) 174 ⟩ Used in section 173.
⟨ Splice independent paths together 890 ⟩ Used in section 887.
⟨ Split off another *rising* cubic for *fin_offset_prep* 504 ⟩ Used in section 503.
⟨ Split the cubic at *t*, and split off another cubic if the derivative crosses back 499 ⟩ Used in section 497.
⟨ Split the cubic between *p* and *q*, if necessary, into cubics associated with single offsets, after which *q* should point to the end of the final such cubic 494 ⟩ Used in section 491.
⟨ Squeal about division by zero 950 ⟩ Used in section 948.
⟨ Stamp all nodes with an octant code, compute the maximum offset, and set *hh* to the node that begins the first octant; **goto** *not_found* if there's a problem 479 ⟩ Used in section 477.
⟨ Start a new row at (*m*, *n*) 1172 ⟩ Used in section 1170.
⟨ Start black at (*m*, *n*) 1170 ⟩ Used in section 1169*.
⟨ Stash an independent *cur_exp* into a big node 829 ⟩ Used in section 827.
⟨ Stop black at (*m*, *n*) 1171 ⟩ Used in section 1169*.
⟨ Store a list of font dimensions 1115 ⟩ Used in section 1106.
⟨ Store a list of header bytes 1114 ⟩ Used in section 1106.
⟨ Store a list of ligature/kern steps 1107 ⟩ Used in section 1106.
⟨ Store the width information for character code *c* 1099 ⟩ Used in section 1070.
⟨ Subdivide all cubics between *p* and *q* so that the results travel toward the first quadrant; but **return** or **goto** *continue* if the cubic from *p* to *q* was dead 413 ⟩ Used in section 406.
⟨ Subdivide for a new level of intersection 559 ⟩ Used in section 556.
⟨ Subdivide the cubic a second time with respect to x' 412 ⟩ Used in section 411.
⟨ Subdivide the cubic a second time with respect to $x' - y'$ 425 ⟩ Used in section 424.
⟨ Subdivide the cubic a second time with respect to y' 416 ⟩ Used in section 415.
⟨ Subdivide the cubic between *p* and *q* so that the results travel toward the first octant 420 ⟩ Used in section 419.
⟨ Subdivide the cubic between *p* and *q* so that the results travel toward the right halfplane 407 ⟩ Used in section 406.
⟨ Subdivide the cubic with respect to x', possibly twice 411 ⟩ Used in section 407.
⟨ Subdivide the cubic with respect to $x' - y'$, possibly twice 424 ⟩ Used in section 420.
⟨ Subdivide the cubic with respect to y', possibly twice 415 ⟩ Used in section 413.
⟨ Substitute for *cur_sym*, if it's on the *subst_list* 686 ⟩ Used in section 685.
⟨ Substitute new dependencies in place of *p* 818 ⟩ Used in section 815.
⟨ Substitute new proto-dependencies in place of *p* 819 ⟩ Used in section 815.
⟨ Subtract angle *z* from (*x*, *y*) 147 ⟩ Used in section 145.
⟨ Supply diagnostic information, if requested 825 ⟩ Used in section 823.
⟨ Swap the *x* and *y* coordinates of the cubic between *p* and *q* 423 ⟩ Used in section 420.
⟨ Switch to the right subinterval 318 ⟩ Used in section 317.
⟨ Tell the user what has run away and try to recover 663 ⟩ Used in section 661.
⟨ Terminate the current conditional and skip to **fi** 751 ⟩ Used in section 707.
⟨ The arithmetic progression has ended 761 ⟩ Used in section 760.
⟨ Trace the current assignment 998 ⟩ Used in section 996.
⟨ Trace the current binary operation 924 ⟩ Used in section 922.
⟨ Trace the current equation 997 ⟩ Used in section 995.
⟨ Trace the current unary operation 902 ⟩ Used in section 898.
⟨ Trace the fraction multiplication 945 ⟩ Used in section 944.
⟨ Trace the start of a loop 762 ⟩ Used in section 760.
⟨ Transfer moves dually from the *move* array to *env_move* 520 ⟩ Used in section 518.
⟨ Transfer moves from the *move* array to *env_move* 514 ⟩ Used in section 512.
⟨ Transform a known big node 970 ⟩ Used in section 966.

⟨ Transform an unknown big node and **return** 967 ⟩ Used in section 966.
⟨ Transform known by known 973 ⟩ Used in section 970.
⟨ Transform the skewed coordinates 444 ⟩ Used in section 440.
⟨ Transform the x coordinates 436 ⟩ Used in section 433.
⟨ Transform the y coordinates 439 ⟩ Used in section 433.
⟨ Treat special case of length 1 and **goto** *found* 206 ⟩ Used in section 205.
⟨ Truncate the values of all coordinates that exceed *max_allowed*, and stamp segment numbers in each *left_type* field 404 ⟩ Used in section 402.
⟨ Try to allocate within node p and its physical successors, and **goto** *found* if allocation was possible 169* ⟩ Used in section 167.
⟨ Try to get a different log file name 789 ⟩ Used in section 788*.
⟨ Types in the outer block 18, 24*, 37, 101, 105, 106, 156*, 186, 565, 571, 627, 1151 ⟩ Used in section 4.
⟨ Undump a few more things and the closing check word 1199* ⟩ Used in section 1187.
⟨ Undump constants for consistency check 1191* ⟩ Used in section 1187.
⟨ Undump the dynamic memory 1195 ⟩ Used in section 1187.
⟨ Undump the string pool 1193 ⟩ Used in section 1187.
⟨ Undump the table of equivalents and the hash table 1197 ⟩ Used in section 1187.
⟨ Update the max/min amounts 351 ⟩ Used in section 349.
⟨ Use bisection to find the crossing point, if one exists 392 ⟩ Used in section 391.
⟨ Wind up the *paint_row* parameter calculation by inserting the final transition; **goto** *done* if no painting is needed 584 ⟩ Used in section 582.
⟨ Worry about bad statement 990 ⟩ Used in section 989.